LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-TR-662004

Steering Languages and Future Science Codes

L. E. Busby

October 3, 2014

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

Steering Languages and Future Science Codes

Lee Busby, LLNL
busby1@llnl.gov

September 15, 2014
Rev. 11

Contents

List of Tables

Steering Languages and Future Science Codes

Lee Busby, LLNL
busby1@llnl.gov

1. Summary Recommendations

- Build a pilot code using *Lua* as the steering language.[1] Standard Lua was 3.5× smaller and 10× faster than Standard *Python* across five Scimark benchmarks. It may be more suitable for static linking and as an *embedded* steering language. It is mature, well-documented, and widely used in many areas, with a recent uptick in scientific users and codes. These points and others will be more fully developed in the sections that follow.
- Use Python more, not less. Python is a very good choice as a steering language. Moreover, it has become an excellent choice as a *primary* programming language for many application areas and uses. There are a wide variety of tools and techniques now that effectively allow the scientist/programmer to write (just) Python, then selectively choose how and how much to optimize their code as needed for speed. Consider purchasing support for a comprehensive Python user environment such as Anaconda [CON1] for a period of time.
- Consider some lessons from three components of the *Basis* project, and try to carry some of their good ideas into future work:
  A) The Basis run-time database (RDB), which is shared with and fully accessible from compiled code, using an ISO C API;
  B) A data description language (DDL) that developers use to define and initialize the RDB;
  C) A domain specific language (DSL) for use in building the user interface for each project.

The first two bullets taken together are not intended to suggest an either/or future. Python and Lua can work together well if and when that becomes desireable. However, the interfaces between codes will continue to be important. Interfaces that emphasize native types will usually be a good choice if language interoperability is important.

2. Structure of the Paper

The rest of the paper is in five main parts. Some general background information and terminology is given in the next section. This will hopefully make it easier to follow the discussion in the remainder. Each of the three recommendations above will be discussed

[1]Disclaimer: The author is participating in an ongoing evaluation of Lua as a steering language for the *Blast* [KOL1] code.

separately, with a final section for concluding remarks and acknowledgements. Three appendices contain information to extend or complete earlier material.

3. What is a Steering Language? Why Use One?

3.1. Background

In 1984, Paul Dubois [DUB1] introduced *Basis* and the idea of *steering languages* to LLNL. His work preceded *Perl* (1987), *Tcl* (1988), and *Python* (1989) by some years. *MATLAB* [MOL1] the company was also founded in 1984. Basis had some similarities with MATLAB, but it went much further in defining ways to connect an interactive command interpreter to compiled code packages. John Ousterhout's 1990 paper [OUS1] introduced the idea of a language-in-a-library (*Tcl*), embeddable in other tools, to many people outside our scientific community. He went on in 1998 [OUS2] to divide (most) programming languages into two main categories: Weakly typed *scripting* languages like Tcl, used on their own for rapid development, or to ‘‘glue’’ together components written in strongly typed *system* languages like C. He suggested that ‘‘*Scripting and system programming are symbiotic*’’, and, to paraphrase, that scripting languages allowed many more, and more casual programmers, to make effective use of computing power.
Python hardly needs further introduction now. *Lua* [LUA1], [IER1] is less well-known. It began in 1993 as a configuration language, mostly for industrial clients of *Tecgraf*, the Computer Graphics Technology Group of PUC-Rio in Brazil. It slowly evolved into a complete language over the next 7-10 years. In a 2011 paper, Ierusalimschy et al. [IER2] distinguish two ways that a scripting language can be integrated with code written in a system language: In the first form − *extending* − the main program is written in the scripting language, which is extended using libraries and functions written in the system language. In the second form − *embedding* − the main program is written in the system language, now called the *host* program, which can run scripts and call functions defined in the scripting language. Although this terminology is not precise, (most real codes have some of both types of integration), I will use their definitions for those terms in the rest of the paper. Also, the term *steering language* implies a context where two languages are being used together, one to ‘‘steer’’ the other, whereas *scripting language* may be just the one. It's often fine to write complete programs using (only) Python, Lua, or Basis. I will use those terms synonymously, unless the context requires otherwise.

3.2. Your Point of View

Ok, let's test your knowledge so far with a short quiz. *The proper form for a next generation physics code is:*

A) An ensemble of mostly independent compiled packages, glued together with and *extending* a central steering language that ‘‘knows all, sees all’’;
B) A coherent compiled core, composed of packages designed to know about one another and work together well, along with an *embedded* scripting language used for problem setup and such other tasks as found to be necessary;
C) Sometimes one, sometimes the other − I just do not know.

(My own answer is probably C, by way of A.) The small questionnaire that I passed around in April was partly for fun, to introduce my study, but it did make clear that otherwise reasonable people have different opinions. Here is the split between AX− and B−division responses to a couple of the questions posed then:

Question				**AX**	**B**
The steering language is →	Very Important	**:**	So So	16:3	3:3
Python is the obvious choice →	Yes! Python forever	**:**	Well, okay	12:5	2:4

Table 1 − **Two Questions for AX and B**

That is hardly significant in any statistical sense, of course, and if my sample failed to account for your opinions, I am sorry. However, our differences, such as they may be, are not just confined to LLNL. Game developer Tim Sweeney [SWE1] identified four reasons why mixing a scripting language into compiled code might create problems, which are paraphrased here:

- [As a system grows] there is increasing pressure to expose more of its native C++ features to the scripting environment. [The system] eventually grows into a desert of complexity and duplication.
- There is a seemingly exponential increase in the cost and complexity [of the] “interop” layer where C++ and script code communicate. [Interop] becomes very tricky for advanced data types such as containers
- Developers seeking to take advantage of [...] native C++ features end up dividing their code unnaturally between the script world and the C++ world
- Developers need to look at program behavior holistically, but quickly find that script debugging tools and C++ debugging tools are separate and incompatible.

Sweeney and his team felt strongly enough about these issues to rip the existing (proprietary) scripting language back out of their popular C++ game engine, after putting it in and living with it for some years. Although I feel that scripting languages do bring net benefit to our physics codes, I acknowledge Sweeney’s points, and allow that I have encountered them in the past. The questionnaire also asked you to rate the relative importance of several ‘‘use cases’’ for a steering language. Here are the average scores from 25 replies. (In this case, AX− and B−division responses were lumped together.)

Use Case	**Mean Score(L=1,M=2,H=3)**
Set up job input, check syntax	2.70
Postprocess output	2.61
Make custom output files	2.50
Steer code during run	2.44
Play around, learn code	1.78
Handle unusual input, etc.	1.86

Table 2 − **Some Steering Language Use Cases**

Apparently we’re a little uncomfortable with the idea of playing around at work, even to learn how to use a new code. There isn’t a lot more to say about those specific results, but defining a reasonable set of use cases is certainly part of a successful steering language project. One additional table (3) shows how the responders self-identified as a ‘‘designer’’ or as a ‘‘developer’’.

I am more of a ...	AX	B
Designer **:** Developer	10:10	4:2

Table 3 − **Self-identification by Division**

Finally, in B-division, the breakdown by years in field (<10:10−20:>20) was 1:3:2; AX was 6:4:9.

4. Recommendation: Build a Pilot Code Using Lua

The primary purpose of this section is to draw out the differences between Python and Lua, in order to justify the recommendation. That said, the languages have quite a bit in common. Their syntax is generally procedural. Assignments, function definitions and function invocations are easy to recognize in either language. Python uses indentation to indicate block structure. Lua is more traditional in its use of keywords to indicate block structure. White space in Lua is freeform. Lua is simple, but it has some sophisticated features such as first-class functions, closures, coroutines, iterators, and more. (None of which a casual user needs to understand.) Syntactically, it seems fair to say that neither language requires much effort to learn and use. Unlike Python, Lua has no pre-defined *class* statement, but it does have mechanisms that allow object-oriented interfaces to be constructed, if desired. At the level of linking to C code, the Lua API contained about 113 functions in 2007, whereas the Python API had about 656 public functions. [MUH1] (Neither has changed greatly in the last seven years.) A short example below illustrates basic syntax. More example code is available in the Scimark benchmarks run as part of this study. [LEB1]

Python:

```
def factorial(n):
  if n == 0:
    return 1
  else:
    return n*factorial(n-1)
```

Lua:

```
function factorial(n)
  if n == 0 then
    return 1
  else
    return n*factorial(n-1)
  end
end
```

There are at least two major implementations of the Lua library and language. The first is ‘‘Standard Lua’’, the original version from Brazil. The second is *Luajit*, [LUJ1] which is a just-in-time compiler for Lua. To a first approximation, the two versions are compatible, and compatible at the application binary interface (ABI) level: A code can link to either library without recompiling or any special effort. Luajit is probably responsible for much of the recent surge in interest among scientific users of Lua. Besides being much faster in general, Luajit has an excellent foreign function interface that can simplify much of the effort in linking to C libraries. For us at LLNL, however, it's important to note that Luajit is not presently available for the PPC64 architecture (Sequoia). Standard Lua is famous for its portability, and is available for all our platforms.

4.1. *Comparison of Source Code Size*

Earlier, the comment was made that Lua is smaller than Python. That is true in several senses. Let's begin with the overall size of the source code for the respective packages. The following table gives some basic metrics about Lua, Python, and several related or otherwise interesting codes. LLVM is included because it is required for Terra, Numba, and Julia, which will be discussed later. The numbers reported in the table are in thousands of lines of code (KLOC), and were collected using the Perl script *cloc*. [CLO1] Blank lines and comments are not counted.

Code-Version	C/C++	Python	Lua	Other	Total	Dependencies
Lua-5.1.5	12.7	-	0.4	-	13.1	C
Luajit-2.0.3	59.6	-	8.9	-	68.5	C
LLVM-3.3	1465	11.6	-	369	1846	C, C++
Terra-6768359	8.5	-	4.3	-	12.8	LLVM, Luajit
Python-2.7.2	426.8	405.2	-	45.0	877	C
Numpy-1.6	107.7	75.9	-	-	183.6	C, Python
Scipy-0.9	438.5	76.8	-	-	515.3	C, C++, Fortran, ...
Cython-0.20.2	8.9	83.4	-	-	92.3	C, Python
Pypy-2.3.1	48.4	1002	-	5.2	1056	C, Python, ...
Numba-0.13.3	8.7	66.2	-	-	74.9	LLVM, Numpy, ...

Table 4 − **Code Sizes for Some Language Implementations**

We need to be careful in touting how small something is, of course. Small size is false economy if you give up required functionality. Python + Numpy + Scipy is an indispensable tool for many scientists. Together, they sum to more than 1½ million lines of code. Is that required for basic code steering? In my opinion, Standard Lua (13 KLOC) or Luajit (69 KLOC) are capable of handling the basic steering task, and enough smaller as to make a qualitative difference in planning for their support and maintenance within a development team.

4.2. *Speed and Memory Benchmarks*

We can argue about functionality, but lines of code is fairly objective and easy to measure. Demonstrating differences in memory size and execution speed takes more work. To that end, I ran two separate sets of benchmarks that included the languages of interest.

Expression Parsers

Expression parsers are a little byway in programming that I learned a bit more about in doing this study. Here at LLNL, the *Blast* code uses an expression parser named *Fparser*, which allows users to define at runtime what are effectively one-line functions of 2−4 real variables. We duplicated the functionality using Lua recently, and were curious about the relative performance. It's an area of steering language use that I had not had much experience with before, so I carried out a set of benchmark runs, the mean results of which are summarized in the short table here. Arash Partow [PAR1] has done a much more extensive set of benchmarks for expression parsers, but did not include Lua or Python in his tests. Note that expression parsing in the sense of these tests is inherently scalar. In particular, it would be *possible* to run the tests using Python + Numpy, but that would run slower (I checked) than standard Python, because the tests do not use vectors.

Language →	C++(sec)	Fparser	Lua	Luajit	Python
Total seconds | Mean Ratio →	15.5336	1.51	4.39	1.70	10.26

Table 5 − **Summary Expression Parser Benchmark Results**

Full results of my runs are given in Appendix A. The complete code to reproduce these benchmarks and the following Scimark results is at [LEB1] . As seen in the table, pure C++ averages about 1½ times faster than Fparser over 26 various expressions, and is about 10 times faster than Python over the same set. These results are much better for all three scripting languages than most general benchmarks demonstrate. I believe that is due to the fact that expressions, or at least these 26, tend to concentrate most of their effort in standard math library routines. All five of the languages link to the same math library, and computing $sin(a) + cos(b)$ will spend most of its time in the library, regardless of which language is being used.

Scimark Benchmarks

The *Scimark* [SCI1] benchmarks in their original form include Java and C implementations of five common numeric algorithms. M. Pall [PAL1] wrote a lua/luajit version of the tests. H. Ardo [ARD1] wrote a Python/Pypy version of Scimark, as part of a larger set of benchmark tests for the *Pypy* [PYP1] project. I used Pall's code without any essential changes for the Lua, Luajit, and Terra results given here. I modified his code slightly for the results labeled ‘‘lj-uopt’’: Luajit has the ability to create ‘‘unboxed’’ arrays of native C types, and it has bit operators distinct from standard Lua. The lj-uopt results turn OFF those optimizations.

I modified Ardo's code as needed to make a standalone version of Scimark for Python. That code is used by both standard Python and Pypy. The Numpy, Cython, and Numba versions each required their own separate set of modifications. The reader may note that I am not a Python expert, nor Lua, nor any other of the scripting languages represented here, for that matter. The code that I wrote is intended to be representative of ‘‘a good first attempt’’ in each language. Many possible optimizations could no doubt be made. (I would be happy to receive corrections and improvements.) Furthermore, some of the code ‘‘cheats’’: For example, the Numpy version uses its builtin random number generator instead of the version from Ardo, because any reasonable Numpy programmer would do that. And a real Cython programmer would probably use Numpy to do more of the work in that version, but I chose to keep most of the loops in Cython, to better understand what that implementation by itself can do.

The Scimark benchmarks run five separate tests, in order: A fast fourier transform (FFT), Successive over-relaxation (SOR), a Monte Carlo calculation of the value of π (MC), a

Raw Scimark results in MFLOPS are shown for cc-O2. Values for other languages are in ratio to the cc-O2 values: A smaller number is faster.

LANG	FFT	SOR	MC	SMV	LU	MEAN	PKMEM(MB)
cc-O2	714.9	802.9	201.8	1032.1	1422.2	834.8	0.27
luajit	1.78	0.87	0.81	2.59	1.17	1.31	0.85
lj-uopt	1.82	0.90	2.20	3.72	1.56	1.63	1.0
lua	61.9	26.2	24.1	55.7	54.3	43.8	1.1
terra	1.55	0.88	0.80	2.52	1.18	1.29	7.2
python	276.	1041.	200.	232.	1883.	436.	3.9
numpy	3.23	2.33	230.	2.31	1.00	1.71	29.1
cython	12.3	1.41	6.23	3.23	7.19	3.55	269.4
pypy	5.44	1.45	4.04	8.25	7.07	3.93	32.6
numba	1.99	1.03	38.1	2.02	869.	2.53	110.0

Table 6 − **Scimark Results: Ratios to cc-O2 Values, plus Peak Memory(MB)**

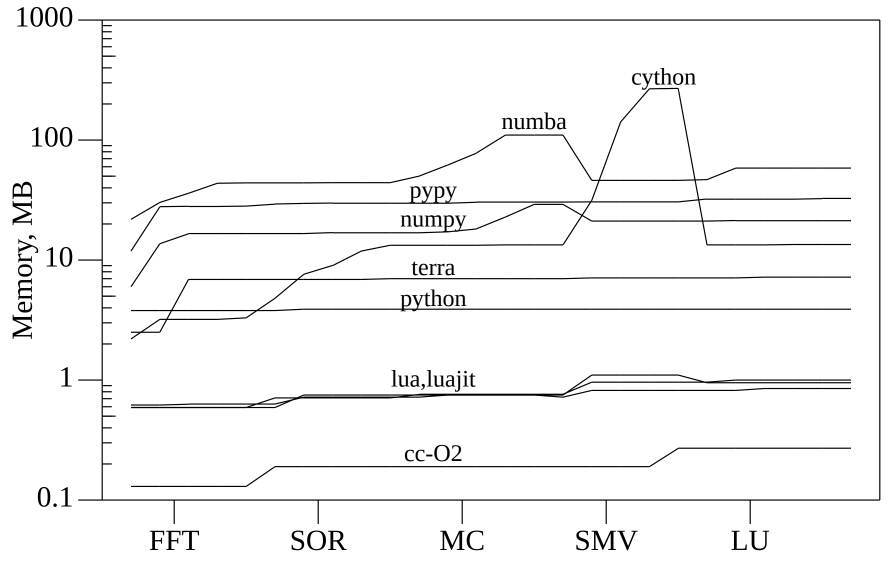

Fig. 1: Memory usage during five Scimark tests

sparse matrix−vector multiplication (SMV), and an LU decomposition of a matrix (LU). The original benchmarks run ‘‘small’’ and ‘‘large’’ tests, in the sense of memory; I ran only the ‘‘small’’ versions. Specifically, each problem is set up at the given size, then timed for running $N = 1$ cycle. If the cpu time required to do that exceeded 2.0 seconds, the test is finished. Otherwise, N is doubled and the test runs again, until cpu time is greater than 2 seconds. A simple formula is then applied to compute the number of floating point operations required to do the task, divided by the cpu time, to output a MFLOPS rating for that test. In the table, raw results (‘‘MFLOPS’’) are shown for the ‘‘cc-O2’’ row. (cc-O2 is the C version of Scimark, compiled at the -O2 level.) Results for the other languages are given as a ratio to those values. So for any language except cc-

O2, a smaller score is better. For example, Lua was 61.9 times slower than cc-O2 running the FFT test, and Lua's raw Scimark score for that test was $714.9 \div 61.9 = 11.55$. Scores reported in the *MEAN* column are the ratio of the mean Scimark scores, not the mean of the ratios.

Each language required about 30 seconds to run the full set of five tests. While the test was running, I separately measured the memory size of the test process, at one second intervals, using the *smem* [MAC1] program. Peak memory usage, in megabytes, is reported in the ''PKMEM'' column of the table. Those memory results seemed interesting enough to plot as a full time series, seen in Figure 1. The x-axis measures time, with marks showing roughly which of the five tests was running at a given moment. (It's difficult to be precise, because the clock time varies slightly among the 10 languages.) Note that the y-axis is logarithmic.

The Scimark and memory usage results presented here are broadly consistent with other benchmark tests that I have seen, so I was not especially surprised by them. They are, I believe, consistent with the earlier assertion that Lua and Luajit are ''noticeably faster and smaller''. Standard Lua was about 10× (436÷43.8) faster and 3.5× (3.9÷1.1) smaller than standard Python. That said, these benchmarks, like all others, mostly demonstrate how difficult it is to produce definitive results. I spent almost two working weeks assembling all the executables onto one machine, writing code, debugging, and optimizing in an effort to produce results that are, in the words of a famous news network, fair and balanced. Two weeks was barely enough. Some other interesting Python implementations include *Parakeet*, *Pyston*, *Theano*, and *Pythran*. Time considerations limited me to the five shown here. A few more comments about the specific results and my experience:

- Numpy gave the best overall performance among the five Python implementations tested, by a fairly substantial margin. I was disappointed with its results on the Monte Carlo test; surely that can be improved. Also, in the SOR test, the natural way to process the matrix using Numpy is to use an array expression to compute each element as the average of its four neighbors. Array expressions create temporary copies, so would appear to require a non-progressive algorithm. In short, the Numpy version of the SOR test is actually using Jacobi iteration, which requires $\omega < 1$ for convergence. Jacobi is technically slower than SOR, but in this case the array expression is still at least 100 times faster than the unvectorized loops − any self-respecting Numpy programmer would do it that way, I think.
- Cython is a hybrid language, mostly Python, with type declarations that look familiar to a C programmer. Cython gave very respectable results here, although its memory usage was quite high towards the end. Cython seems to balance the several needs of a scientific programmer rather well. It allows optimization to be done a little at a time, it's friendly to Numpy, it's easy to link to other libraries, the syntax seems clear to a C programmer, and it felt stable. On this particular benchmark, Cython could easily have had the best score among the five Python implementations tested: It could have run the Numpy code in every case except Monte Carlo, where the unvectorized Cython code was 37× faster than (vectorized) Numpy.
- Pypy is a just-in-time implementation of Python. Pypy is the only other code capable of

running exactly the same script as standard Python (just 111× faster.) It therefore didn't require any fiddling on my part. Unfortunately, Pypy and Numpy/Scipy seem not to get along very well.

- Numba turned in the second best overall speed results among the Python implementations. However, it seemed ... touchy. And the results for the LU test are just mysterious to me. The code for Numba looks nearly the same as the Pypy version, except for the ''@jit'' decorator. Yet Pypy ran that test more than 122 times faster than Numba. I tried several variations of the LU code for Numba over several days, and found nothing that would improve the reported results. Surely this is a case of user ignorance or error.

Crosscheck

Two LLNL codes − Lasnex and Ares − that optionally load Python as a library were available to me.[2] This made it possible to measure memory size of the running executable before and after loading Python. Lasnex loads Python dynamically. Immediately after starting, Lasnex had a memory footprint of about 38.6MB. Starting Python (including Numpy) changed that to 49.3, an increase of about 10.7MB. Ares loads Python statically. The non-Python version was about 28.4MB. Linking Python gave a starting size of 35.6; initializing and importing Numpy and Scipy increased memory to 50.7MB. So in very rough terms, adding Python and Numpy to a code appears to add anywhere from 10 to 22MB or so to the size of the memory image. Measurement of a standalone copy of Python showed an initial size of 3.2MB; importing Numpy increased the size to 11.7MB. These are similar numbers to the values seen during the Scimark tests.

5. Recommendation: Use Python More, not Less

Enthusiasm for Python as a steering language has perhaps only been exceeded by enthusiasm for it as a *scripting* language. That is, many people [LUC1] would be happy to use only Python, and at least figuratively, leave the system language to library authors. Much of the more recent work in the Python development community reflects that sentiment: Pyrex, Cython, IPython, Pypy, Numba, Pyston, Parakeet, and many others are efforts to make Python the scripting language into a faster and therefore more comprehensive tool.

My own informal survey of recent scientific Python projects suggests to me that the center of gravity has indeed shifted. The standard tools once were SWIG and Numpy, followed shortly by Scipy. It seems more common now to write scientific programs using Cython. In addition to excellent support for Numpy and Scipy, Cython's foreign function interface can obviate much of the effort of wrapping existing system language code. It is highly compatible with standard Python, and my own tests showed that simply compiling an otherwise unchanged Python module with Cython resulted in a dynamically loadable module about 2× faster than standard Python. From there, optimization can proceed a little at a time. The modules produced by Cython are not small, and the running code may take substantial memory, as was demonstrated earlier in the Scimark runs, but that is really not an issue most of the time.

As a ''desktop interface'', IPython has good support for Cython, along with Pypy,

[2]These measurements were made on an X86_64 (64 bit) system.

Numba, and many other of the various analytic and plotting packages for Python. The one difficulty that I experienced during the preparation of my study was in assembling a complete, self-consistent Python environment that included all the pieces for doing the work. My own job assignment over most of the past 15-20 years has not required much Python work. I developed a couple of fairly extensive Python modules in 1996-97, but would otherwise describe myself as an occasional user. It has never been trivial to assemble all the parts of a substantial Python environment, and the current plethora of tools makes that harder, not easier.

My work benefited greatly from the availability of the ‘‘Anaconda’’ Python distribution [CON1] from Continuum Analytics. I was fortunate that my primary machine for running the benchmarks is supported by an existing binary distribution of Anaconda. Given the increased interest in Python as a scientific end-user language, and given the difficulty in building a complete environment, I specifically recommend that we consider purchasing support for a distribution such as Anaconda for a period of time. There are other similar Python distributions available, and I defer to others more knowledgeable in selecting a particular one.

It is also important to distinguish here between Python as a scripting language and Python as a steering language. As a scripting language, our entire community of Python programmers and users would benefit from a relatively standard, complete, and up-to-date environment. As a steering language, the several developer groups responsible for code projects have several sets of requirements and constraints. They may well choose to customize Python as necessary for their particular code, and should continue to be able to do so.

6. Recommendation: Consider Some Ideas from The Basis System

The *Basis System* [DUB2] started as part of the MERTH project in the spring of 1984, in the Magnetic Fusion Energy (MFE) program. The first usable code began to appear in the winter of 1984-85. Basis spread through MFE for the next 3-4 years, then was chosen as the computer science infrastructure for the Unix port of the *Lasnex* [ZIM1] code. The Lasnex/Unix project required another 3-4 years to mostly complete, followed by 6-8 years of continuing refinement of Basis, which continues at a low level up to the present time.

Basis itself was initially written in the Basis dialect of Fortran, called *MPPL*. MPPL is a close cousin of the Unix *m4* macro processor, with slightly more conventional syntax and some features specific to Fortran (77), such as line length and column conventions. Over the years, most of Basis has been converted to C. Only one package remains in MPPL today, for testing purposes. Lee Taylor has, within the last 5 years or so, added very good support for modern Fortran in Basis.

Basis was an ambitious project, [DUB3] with a lot of components that each pushed the boundaries of language and user interface design and integration. It is still unique in the sense that all its components were *designed* to work together as part of a steering language system.

6.1. The Basis Run Time Database

Basis defines and creates a *run-time database* as one component. Consider one element of the Basis RDB: It has a complete C-compatible API for use from compiled code (the *host program*.) Most operations on the RDB that can be done through the Basis interpreter can also be done by the host program using that API.[3] A more modern name for this capability is *introspection* − the host program can inspect its own variables and functions without going to the scripting language. Thus operations such as dump/restart, mesh modifications, and so forth can be accomplished entirely from compiled code, using standard calls to interact with the RDB. This is simple and efficient. Making it depend on the scripting language would be worse, not better.

6.2. The Basis Data Description Language

The run-time database can obviously be modified dynamically. However, much of it is actually constructed and initialized at compile-time, using a data description language (DDL) that Basis called a ‘‘variable descriptor file’’ (VDF). Programs are broken into packages, each of which is described by one or more VDF's. The information in a VDF is similar to ‘‘header’’ files: Variable types, shapes, sizes, function signatures, ‘‘groups’’ that collect together those things, macro definitions, parameters, documentation, comments, etc.[4] It is prosaic and from today's perspective, even a little passé.[5] Aspects of the VDF that need to be remembered are about as follows: Static initialization of the RDB is quite important. The Lasnex RDB has perhaps 1200 entries, but any given problem input file touches only a tiny fraction of those variables. The default values for the rest were set in a VDF. Default values themselves change as time goes by. Lasnex also has a system to track default values through time, so that if an old problem is restarted, variables receive the default value that was in effect at the date the problem ran, instead of the date of the code that is running it.

The VDF was designed to balance the needs of a human reader/developer against the needs of a computer parsing the file. In general, the VDF favors the human. Most of the information about a given object is gathered together in one spot, and is organized to give the human maximum clarity with minimum effort. Header files, plus optional directives that guide a program such as *SWIG* in generating ‘‘glue’’ code accomplish the same general ends as does a VDF. But the VDF was *designed* to do its job, and the developers for whom it was specifically designed, do notice. This principle can be carried into a future data description language.

6.3. The Basis User Interface Construction System

One last component in Basis seems even more of an anachronism: The Basis scripting language contains its own macro processor. This can be used to define a domain specific language (DSL) as the user interface for a given project. Even as macro processors go, (Dubois was famous for his ‘‘I hate macros’’ comments) the Basis version is painfully complicated to work with. Nevertheless, many users consider it a gift.[6] I don't personally

[3]This feature − a C API that gives the host program full access to the Lua state − is cited by the Lua authors as one of the key reasons for that language's success, and is an enduring design goal

recommend a macro processor. But as an element of user interface design, the Basis experience is worth remembering. People do appreciate a user interface that is concise and designed for the task at hand. Although a steering language does sometimes require power and generality, the interface designed on top of it also needs to be as simple as possible. Several users and developers commented to the effect that a ‘‘program input deck should not be an API.’’

One other advantage that Basis had, not easily carried forward, is that its scripting language is essentially equivalent to its systems language (Fortran.)[7] This was certainly part of the design: Early Basis documents suggest that algorithm development could be done using the scripting language, then easily transferred to MPPL/Fortran with little change. It is difficult to even properly appreciate this feature now. Language design has moved on, and we imagine that it’s a good thing for the scripting language to be independent from the system language.[8] Still, when you consider the effort that has gone into, say, Pyrex and Cython, to achieve what is basically a blending of the scripting and system layers, it is useful to remember that easier solutions to a similar problem have been found in the past.

6.4. Basis Summary

This has been a long section. To reiterate the main points, here is a summary of some of Basis’ good ideas:

- An run-time database (RDB) with a complete ISO C API for the host program(s);
- A language and process for initializing and constructing the RDB, designed for humans; (and, seriously, consider the question of units.)
- A user interface that goes well beyond ‘‘wrapping’’ the internal objects of the compiled code.

7. Conclusion

I considered subtitling this paper *Possibly the Least Surprising Paper of 2014*, then

for their team.

[4]One other item that the VDF formally defines, but never used in practice, is *units*. It is a regret often expressed and deeply felt by Basis users and developers alike that *units* were optional in the VDF. It would not be trivial to require units in a future database, but it is certainly worth considering.

[5]In 1984, Fortran didn’t have header files; it was necessary to invent the VDF from the ground up.

[6]Interestingly, it seems that Lua has considered a macro processor as late as 2007. In [IER1] , §7, the comment is made that ‘‘*We still have not completely dismissed the idea of providing Lua with a macro system: it would give Lua extensible syntax to go with extensible semantics.*’’ (But don’t hold your breath....)

[7]This included strong typing. The Basis scripting language requires that variables be declared before use, although it provides a ‘‘chameleon’’ type that can vary. There is a ‘‘modern’’ trend towards stronger typing in dynamic languages. It’s good to know that the Wheel of Reincarnation is yet revolving.

[8]I know of some scripting languages that emulate the syntax of C or C++, but none has become generally popular. Leaving aside the technical issues, neither of those languages is likely to win a beauty contest.

decided that (as usual) I was not really taking account of the long term. I *have* learned some new, if mostly unsurprising to me, things. (And I do acknowledge that gathering together unsurprising facts can sometimes be useful, and that not everyone has had my experiences.)

Expression parsers are an interesting small piece of the steering problem, new to me. My attitude towards the general process of wrapping code, especially in the *embedded* case, changed noticeably as a result of reading, thinking, talking to other people, and trying out a few simple things.

More than that, I have come to appreciate some aspects of what I once treated as received knowledge in a slightly different light. In [DUB4] we wrote that ... *the most changeable aspect of a scientific computer program is what users want to calculate with it.* That certainly does sound true. Later on (1998), in [DUB5] Dubois wrote that in his observation, most average scientists and engineers in fact ‘‘program computers for money’’ part of the time, and are therefore to that extent professional programmers. That also may be true, and more so as time passes. Yet I find myself not sure that I entirely agree. More importantly, I’m not sure that you the users of scientific programs agree.

The realistic choices that we have with regard to steering languages today make it all too easy for the wrapping step to also *become* the user interface. That seems more like ‘‘if it’s good for the programmer it’s good for the user’’, which is a step beyond ‘‘setting the scientist free’’, and may well not be true. Whatever else their benefits, steerable codes require a collaboration between programmers and users. Like any collaboration, success requires that each side needs to come a little more than half way to the center. Otherwise the gap in the middle is all too clear.

7.1. Acknowledgements

One last conclusion is that this paper was rather too large a project for one person. Confronting the sheer energy and scope of scripting languages, wrapping technologies, and various programming environments, is plenty enough to make one feel constantly lazy and ignorant. I could not have done nearly so well without important help from many nearby people. I am grateful to Rob Neely for suggesting the project. To all of you who returned comments on my questionnaire, thank you. I am grateful for extended conversations with Lee Taylor, Brian Ryujin, Kyle Chand, Aaron Black, Dean Williams, Walt Nissen, Tom Brunner, Rob Rieben, and several members of the PMESH and Lasnex teams. Mike Collette and Brian Ryujin personally gathered statistics about the Ares code for my benefit. Kyle Chand, Walt Nissen, and Aaron Black did the same for their codes. Dan Laney, Jeff Keasler, and Tanim Islam sent me extended email commentary about steering languages and their current uses. To each of those individuals, thank you. Finally, I am indebted to the several members of the Basis and Lasnex code development teams, and to the many users of those systems, who have taught me most of what I know about steering languages. If mistakes remain in the paper, they are my responsibility, of course.

8. References and Additional Reading

1. [ABA1] The GSL Shell Project: A Luajit wrapper around the Gnu Scientific Library. See http://www.nongnu.org/gsl-shell/
2. [ARD1] Scimark benchmark for Pypy: See https://bitbucket.org/pypy/benchmarks/commits/04c696b62ec7/
3. [BEC1] Bechtold, Bastian., Lunatic-python, A two-way bridge between Python and Lua. See https://github.com/bastibe/lunatic-python
4. [BEZ1] Bezanson, J., Karpinski, S, Shah, V., and Edelman, A., Julia: A Fast Dynamic Language for Technical Computing, arXiv:1209.5145 [cs.PL], 2012, See http://arxiv.org/abs/1209.5145
5. [BOO1] Boost.python interface generator: See http://www.boost.org/doc/libs/1_55_0/libs/python/doc/
6. [CLO1] Cloc, a Perl script to count lines of code: See http://cloc.sourceforge.net/
7. [CON1] Continuum Analytics, source of Anaconda distribution of Python tools: See http://continuum.io/
8. [CYT1] Behnel, S., et al., Cython: The Best of Both Worlds, Computing in Science Engineering, 2011, v13n2, pages 31-39. See http://cython.org
9. [DEV1] Terra: A low-level counterpart to Lua. See http://terralang.org/
10. [DUB1] P.F. Dubois, A New Architecture for Large Scientific Simulations, 1984, https://e-reports-int.llnl.gov/pdf/197756.pdf
11. [DUB2] P.F. Dubois, et al., *The Basis System*, M-225, Lawrence Livermore Laboratory, Livermore, CA, 1988 (227pp.) Notes about the origin of Basis were taken from the preface of this document. Arguably the ‘‘best’’ Basis manual, it has apparently been lost from, or never entered into, the Lab library. I still keep a paper copy.
12. [DUB3] Dubois, P.F. and Motteler, Z.C., Basis: Setting the Scientist Free, 1988, https://e-reports-int.llnl.gov/pdf/215997.pdf This is a good short summary of the overall Basis system.
13. [DUB4] Dubois, P.F. and Motteler, Z.C., The Basis Code Development System (1995), Lawrence Livermore National Laboratory. See https://e-reports-int.llnl.gov/pdf/226458.pdf
14. [DUB5] Dubois, Paul F., Ten Good Practices in Scientific Programming, UCRL-JC-132268, See https://e-reports-int.llnl.gov/pdf/234862.pdf
15. [FBL1] fblualib: A collection of Lua/Torch utilities. See https://github.com/facebook/fblualib
16. [HOA1] A pair of essays on interactive scientific computing. See http://graydon2.dreamwidth.org/3186.html
17. [IER1] Ierusalimschy, R., de Figueiredo, L. H., Celes, W. 2007. The evolution of Lua. In the Third ACM SIGPLAN Conference on History of Programming Languages: 2.1-2.26, San Diego, CA (June). See http://www.lua.org/doc/hopl.pdf
18. [IER2] Ierusalimschy, R., de Figueiredo, L. H., Celes, W. Passing a language through the eye of a needle, ACM Queue 9 #5 (May 2011) 20-29. See http://queue.acm.org/detail.cfm?id=1983083
19. [KOL1] Blast team members include Tzanio Kolev, Robert Rieben, Veselin Dobrev, Robert Anderson, Michael Kumbera, Thomas Brunner. See https://computation-rnd.llnl.gov/blast/

20.[LEB1] Benchmark code for expression parsing and Scimark tests in this paper: Contact the author, or see https://bitbucket.org/lebusby/splb
21.[LUA1] General starting point for all things Lua: See http://lua.org
22.[LUC1] Lucks, Julius B., Python − All a Scientist Needs, eprint arXiv:0803.1838, 2008. See http://arxiv.org/abs/0803.1838
23.[LUJ1] Luajit project home page: See http://luajit.org/
24.[LUP1] Lupa: Python wrapper around Luajit. See https://pypi.python.org/pypi/lupa
25.[MAC1] Smem memory measurement (python) script: See http://www.selenic.com/smem/ Smem attempts to proportion memory use among the several processes sharing memory pages in a multi-processing operating system All values reported in the paper were taken from the ‘‘PSS’’ output of smem.
26.[MER1] A collection of links to projects that use Lua. See https://sites.google.com/site/marbux/home/where-lua-is-used and see also http://en.wikipedia.org/wiki/Lua_%28programming_language%29 About 100 packages in the Debian repository depend on *liblua*.
27.[MOL1] MATLAB origins story: See http://www.mathworks.com/company/newsletters/articles/the-origins-of-matlab.html
28.[MUH1] Muhammad, H. and Ierusalimschy, R., C APIs in Extension and Extensible Languages, Journal of Universal Computer Science, v13n6 (2007), pps 839-853. See http://www.jucs.org/jucs_13_6/c_apis_in_extension
29.[OUS1] Ousterhout, J.K. (1990). Tcl: An embeddable command language. In Proceedings of the USENIX Winter 1990 Technical Conference, pages 133-146, Berkeley, CA. USENIX Association. See http://web.stanford.edu/˜ouster/cgi-bin/papers/tcl-usenix.pdf.
30.[OUS2] Ousterhout, J.K. (1998). Scripting: Higher-level programming for the 21st century. IEEE Computer, 31(3):23-30. See http://www.tcl.tk/doc/scripting.html
31.[PAL1] Scimark.lua code for tests as run in this paper: See http://luajit.org/download/scimark.lua
32.[PAR1] Extensive tests of several C++ expression parser codes: See http://code.google.com/p/math-parser-benchmark-project/
33.[PYB1] Pybindgen software interface generator: See http://code.google.com/p/pybindgen/
34.[PYP1] Home page for Pypy implementation: See http://pypy.org/
35.[SAT1] Satish, N., et al., Can traditional programming bridge the Ninja performance gap for parallel computing applications?, Proceedings of the 39th Annual International Symposium on Computer Architecture, pages 440-415, IEEE Computer Society Washington, DC, USA (2012). See http://dl.acm.org/citation.cfm?id=2337210
36.[SCI1] Scimark benchmark home page: See http://math.nist.gov/scimark2/
37.[SEL1] Lua interface generator for C++ code: See https://github.com/jeremyong/Selene
38.[SWE1] T. Sweeney comments about scripting language problems: See https://news.ycombinator.com/item?id=7585186 and see also http://tinyurl.com/qcosbz5 for the original posting.
39.[SWI1] SWIG home page: Simplified Wrapper and Interface Generator. See http://www.swig.org/

40.[TOL1] Tolua++ software interface generator for C++: See http://www.codenix.com/˜tolua/

41.[TOR1] Machine learning toolkit featuring Luajit. Collobert, R., et al., Torch7: A Matlab-like Environment for Machine Learning, BigLearn, NIPS Workshop, 2011. See http://torch.ch/ and http://ronan.collobert.com/pub/matos/2011_torch7_nipsw.pdf

42.[ZIM1] Zimmerman, G.B. et al., LASNEX−A 2-D Physics Code for Modeling ICF, Inertial Confinement Fusion, 1996 ICF Annual Report, LLNL, UCRL-LR-105821-96, See https://lasers.llnl.gov/publications/icf_reports/annual_96.pdf

9. Appendix A: Expression Parser Benchmarks

N	C++	Fparser	Lua	Luajit	Python	Expression, as x = f(a,b)
0	0.0367	3.14	76.6	14.8	189.	sin((1.0+2.0/2.0*3.0)*4.0ˆ5)+cos(6.0*π)
1	0.0379	3.89	23.0	14.1	58.0	a+1.0
2	0.0360	4.09	24.2	14.7	59.3	a*2.0
3	0.0385	4.98	24.2	13.9	67.8	2.0*a+1.0
4	0.0382	6.21	25.9	14.1	75.4	(2.0*a+1.0)*3.0
5	0.0421	7.82	64.4	13.4	140.	1.1*aˆ2 + 2.2*bˆ3
6	2.3541	1.28	1.55	1.00	2.68	1.1*aˆ2.01 + 2.2*bˆ3.01
7	1.2282	0.83	2.82	1.07	7.19	1.0/(a*sqrt(2.0*π))*eˆ(-0.5*((b-a)/a)ˆ2)
8	0.0928	7.65	21.1	7.85	72.3	(((((((7.0*a+6.0)*a+5.0)*a+4.0) *a+3.0)*a+2.0)*a+1.0)*a+0.1)
9	0.0837	14.0	102.	8.28	218.	7.0*aˆ7 + 6.0*aˆ6 + 5.0*aˆ5 + 4.0*aˆ4 + 3.0*aˆ3 + 2.0*aˆ2 + 1.0*aˆ1 + 0.1
10	0.0774	4.12	24.7	10.3	57.9	sqrt(aˆ2+bˆ2)
11	0.4761	1.19	3.38	2.01	5.89	sin(a)
12	0.0728	2.32	21.5	9.42	43.2	sqrt(abs(a))
13	0.0379	3.30	31.2	15.2	62.1	abs(a)
14	0.1681	5.55	18.9	5.33	75.5	(a/((((b+(((e*(((((π*((((3.45* ((π+a)+ π))+b)+b)*a))+0.68)+ e)+a)/a))+a)+b))+b)*a)-π))
15	2.1368	1.28	2.52	1.05	6.01	a + (cos(b-sin(2/a*π)) - sin(a-cos(2*b/π))) - b
16	0.9613	1.17	2.81	1.26	4.62	sin(a) + sin(b)
17	0.5437	1.59	6.06	2.58	13.3	abs(sin(sqrt(aˆ2+bˆ2))*255.0)
18	0.1649	2.25	7.63	3.97	25.4	(b+a/b) * (a-b/a)
19	1.1536	1.50	3.02	1.15	7.71	(0.1*a+1.0)*a+1.1-sin(a)-log(a)/a*3.0/4.0
20	1.2522	1.24	2.40	1.10	4.83	sin(2.0 * a) + cos(π / b)
21	1.2731	1.28	2.44	1.08	5.19	1.0 - sin(2.0 * a) + cos(π / b)
22	1.3956	1.33	2.94	1.33	6.46	sqrt(abs(1.0 - sin(2.0 * a) + cos(π / b) / 3.0))
23	0.0806	3.20	12.5	7.38	39.0	1.0-(a/b*0.5)
24	1.6753	1.13	1.76	0.90	3.29	10.0ˆlog(3.0+b)
25	0.0760	2.25	22.4	7.37	42.0	cos(2.41)/b
*	15.5336	1.51	4.39	1.70	10.26	*C++: total seconds; Others: average ratio*

Table 7 − **Full Expression Parser Benchmark Results**

Each expression was evaluated 10 million times. For the C++ column, time in seconds to carry out the evaluation is recorded. For the other four languages, the value recorded is the ratio to the C++ time. The final row records the total C++ time to carry out $26 * 10^7$ evaluations, and the average ratio for each of the other languages. So, compared to C++, Fparser was the fastest of the other languages, about 1.51 times slower than C++. All of these tests were run as *embedded* codes: For each language, a C++ code was built, linking against the language or expression parser library to create an executable program.

10. Appendix B: Wrapping Your Code

‘‘Wrapping code’’ is the term used for creating an interface between a scripting language and some particular set of files written in a system language. The scripting language generally has a C API that defines what it means for the scripting language to call an external (C) function, and vice-versa. Structure definitions, function declarations, macros and so forth are written to allow data to be passed back and forth, and invoke operations between the layers.

In the beginning, wrapping was a programming activity carried out by a human. (And it still can be.) There is often a lot of regularity in the ‘‘glue’’ code that wrapping creates, so we rapidly began to develop tools to automatically write the glue code based on some, hopefully simpler, description of the necessary interfaces.

When the project began, I thought that evaluating wrapping technology and perhaps directly comparing two or three of the tools would be a big part of the work. I did a little of that, and here’s the answer: If you’re interested in wrapping Python, consider SWIG, [SWI1] boost.python, [BOO1] and pybindgen. [PYB1] For Lua, have a look at SWIG, tolua++, [TOL1] and Selene. [SEL1] But before you go off and spend a year, maybe read the rest of this section.

There are many blog postings that begin with words like *Well, I tried SWIG and it didn’t do just what I wanted, so I ended up writing my own* .... I read 6 or 8 of those before I began to realize that there are indeed a lot of ways to interface a scripting language with a system language. Some are general, some are aimed at particular parts of the problem, some write especially scrutable or inscrutable code, etc. Choosing the ‘‘best’’ one is like choosing the best color, only harder, because it’s not so obviously arbitrary. And it really depends on the problem you are trying to solve.

If you are *extending* a code and need to connect lots of functions or methods to your scripting language, SWIG is still pretty good. But a lot of people in that situation nowadays are using the foreign function interface in Cython (or Luajit), and effectively skipping the wrapping step. Be aware that someone on your team needs to understand both sides of the code that is wrapped, and the wrapping technology, and why the choices were made. Understanding, for as long into the future as the code exists, is still necessary, even if the wrapping code is generated automatically.

If you are building an *embedded* code system, think first about the user interface, and how your steering language will work with that. Embedded interfaces tend to exercise the steering language API more rigorously − the host code is doing more of the work − so it can be harder to automate the connection. Hand wrapping can still be a good choice, or can be a good first choice, in order to understand the problems well enough to automate them later.

In an effort to better understand the scope of the problems, I attempted to compile some information about how much wrapping code is actually involved in several code systems. It is difficult to derive these values, and no little judgement is required to say whether any particular line of code is part of the interface between the steering and system languages. Table 8 does demonstrate that interface code can be a substantial fraction of a system.

In the table, I believe the Ares code is the only example of an *embedded* system among

	Interface(KLOC)		Total(KLOC)		
Code Name	System	Scripting	System	Scripting	Wrapping Tool
Ares	10.1(2.2%)	0	456.5	12.6	hand-wrapped
PMESH	81.8(29%)	0	276.5	42.4	SWIG + hand-wrapped
uv-cdat	219.9(40%)	4.3(2.0%)	548.4	215.2	SWIG + hand-wrapped
pygsl+gsl	80.2(26%)	12.7(36%)	314.3	35.5	SWIG
Lasnex	359.8(54%)	44.8(61%)	671	73.9	Custom scripts
KUJO	1.7(37%)	0	4.6	0.3	tolua++

Table 8 − **Interface Code Counts for Several Systems**

the seven. Numbers for the others suggest that ‘‘interface code’’ may be ⅓ to ½ of the total code in a system. This code may be rarely encountered by most of the programmers. However, someone must be capable of understanding the entire system. Few if any of us really enjoy working with SWIG (for example), and that fact is part of the maintenance problem.

11. Appendix C: Comments on Some Current Languages and Programs

There is a great deal of interesting work being done, both in the area of steering languages and outside. In no particular order, here are some comments on current work that may have an impact on our choices in the future.

1. Julia [BEZ1] is an effort to skip the clumsy parts about mixing two languages into one code. It's closer to a steering language than a system language, because it can be easily *extended* using libraries of compiled code. It attempts to have the performance of a system language, using JIT technology. Its dependence on LLVM makes it fairly complex to install and maintain, although Julia proper is still rather small (about 34 KLOC of C, plus 74 KLOC of Lisp.) It can't really be used in an embedded sense, and you can't (presently) use Julia to build object files linkable from another language. But it has many very appealing and innovative features. Hoare [HOA1] called it a ‘‘Goldilocks’’ language. As with the protagonist of that story, I think most of us who care, hope that Julia will succeed.
2. Terra [DEV1] is several kinds of language. As the Scimark results showed earlier in this paper, it is a very high performance implementation of Lua. (It can run unmodified Lua code.) It mixes together Luajit, LLVM, and Clang, so it's not trivial to set it up and make it work. But it works pretty well in my experience so far, given how young the project is.

   The important part of Terra is something quite unexpected. The authors of Satish et al. (2012), [SAT1] showed that optimizing memory cache performance along with small-scale vectorization and threading could improve performance of scientific codes by a factor that averaged over 20×. The paper was very good, and I have no issue with their results. However, I felt the authors were a little disingenous with respect to some of the practical difficulties around memory cache optimization, in particular. It's hard to do in the first place, because cut and try is about the only available strategy. It's even harder to do in the sense of portability. Our current system languages (C, C++, Fortran) limit our ability to easily *find* the optimums, and limit our ability to *express* them in a portable way.

   Terra innovates on both those points. It is a low-level, strongly typed system language that can efficiently capture the machine-level details of optimal code. And that low-level language ‘‘Terra’’ is connected to Lua as a set of first-class objects and mechanisms, so that you can use Lua to efficiently write Terra. For example, you can write loops in Lua to parameterize the search for optimal code in Terra. Furthermore, although Terra has some heavy dependencies as mentioned above, it is designed to create small independent object files as an end product. So it can potentially play well as a small fast part of a code mostly written in some other (system) language.

   From my perspective, Terra is like a 50 pound diamond that fell from the sky, (only with more obvious uses.) Even otherwise sensible people might be a little cautious in approaching. It is definitely worth a look, but not as a steering language.
3. Torch7 [TOR1] is a machine learning toolkit, developed at Idiap Research Institute, New York University and NEC Laboratories America. To my knowledge, it is one of the more complete current scientific systems based on Lua and Luajit. The core of Torch7 is a multi-dimensional array module. It also provides a plotting package, a

binding to the Qt graphics framework, a binding to the Cephes special scientific functions library, and many, many others. There seems to be a team at Facebook developing Torch-related tools [FBL1] and utility libraries.

4. GSL-Shell [ABA1] is a Luajit based wrapper around the Gnu Scientific Library. It can be hard to install from source, because the graphics library it uses is a bit old, but it is very nice once it's running. It is a fine example of the use of the Luajit foreign function interface to easily link to a C library. The GSL is a very good library of general scientific functions, all brought to the Lua command line and made available for easy computation and graphic display.
5. Lunatic-python, [BEC1] Lupa, [LUP1] and one of the libraries cited in the Facebook/Torch7 [FBL1] utilities all aim to connect Lua with Python. No endorsement is made here, only the comment that several packages to carry out such functionality do exist already. Python and Lua each have a complete API to C, so in principle it is entirely reasonable to connect packages in one language with a running interpreter in the other. In practice, that would be relatively simple for simple tasks, but difficult to fully generalize. There are many alternative ways to move data between processes, of course, so it's not clear when or if we would ever wish to do this ourselves.